CHAPTER 3

Functional Programming with JavaScript

There has been quite a buzz lately about functional programming. Purely functional programming languages such as Haskell, Clojure, and Scala are currently being used by tech giants including Google, Facebook, LinkedIn, and Netflix. Popular languages like JavaScript, Python, and Ruby support functional programming techniques although they wouldn't be considered fully functional languages. There is an explosion of libraries in all of these languages that religiously follow and encourage you to follow those techniques. Even traditionally object-oriented languages such as Java and C++ support functional programming with lambdas[1].

If you are wondering where this functional trend came from, the answer is the 1930's, with the invention of lambda calculus[2]. Traditionally, functions have been a part of calculus since it emerged in the 17th century. Functions can be sent to functions as arguments or returned from functions as results. More complex functions called *higher order functions* can manipulate functions themselves and use them as either arguments or results or both. In the 1930's, Alonzo Church was at Princeton messing around with these higher order functions when he invented a universal model of computation called lambda calculus or λ-calculus[3].

In the late 1950's[4], John McCarthy took the concepts derived from λ-calculus and applied them to a new programming language called Lisp. Lisp, which is not purely functional, still belongs to the functional language paradigm in the same way that

1 "A lambda is a block of code that can be passed as an argument to a function call." - http://martinfowler.com/bliki/Lambda.html

2 Lambda Calculus Timeline, http://turing100.acm.org/lambda_calculus_timeline.pdf

3 Lambda Calculus Timeline, http://turing100.acm.org/lambda_calculus_timeline.pdf

4 Lambda Calculus Timeline, http://turing100.acm.org/lambda_calculus_timeline.pdf

JavaScript or Python does. Lisp implemented the concept of higher order functions and functions as *first class members* or first class citizens. A function is considered a first class member when it can be declared as a variable and sent to functions as arguments. These functions can even be returned from functions.

The functional programming trend is not new to computer science. It is a throwback. There is also a chance that you have already written functional JavaScript code without thinking about it. If you've mapped or reduced an array, then you're already on your way to becoming a functional programmer. React, Flux, and Redux all fit within the functional JavaScript paradigm. Understanding the basic concepts of functional programming will make you better at structuring React applications.

In this chapter, we are going to cover how to implement functional techniques with JavaScript, as well as how those techniques have inspired React and Flux. We will wrap this chapter up with introducing the Flux pattern, a design architecture that has sprung from these

What it means to be Functional

A great debate to get into with other engineers is whether or not languages that support functional programming techniques can be called "functional languages." In order to be a functional programmer, you're going to have to take a side in this debate. Are languages that support functional programming alongside other paradigms considered functional languages?

Purely functional programming languages such as Haskell or Clojure strictly enforce the functional paradigm. Other languages, such as JavaScript and Python support functional techniques, but they also support other programming paradigms as well. Many engineers only define a language that enforce the functional paradigm as a "functional language".

The same debate is true for object orientation. JavaScript is not a traditional object-oriented language like C++ or Java. Yet, JavaScript supports objects and inheritance so we've been able to incorporate object-oriented design patterns such as MVC into our applications. Although JavaScript is not an object-oriented language, JavaScript supports object orientation with objects and prototypal inheritance. ES5 beefed up this support by introducing Object.create and Object.defineProperties. ES6 takes it a little further by introducing classes.

JavaScript supports functional programming because JavaScript functions are first class citizens, meaning that functions can do the same things that variables can do. ES6 adds language improvements that can beef up your functional programming techniques including arrow functions, promises, and the spread operator.

We call JavaScript a functional language because it supports first class members, but what does it mean to be first class? It means that functions are variables; they too can represent data in your application. You may have noticed that you can declare functions with the var keyword the same way you can declare strings, numbers, or any other variable.

```
var log = function(message) {
  console.log(message)
};

log("In JavaScript functions are variables")

// In JavaScript, functions are variables.
```

With ES6, we can write the same function using an arrow function. Functional programmers write a lot of small functions, and the arrow function makes that much easier. Both of these statements do the same thing, they store a function in a variable called log. Additionally, the const keyword was used to declare the second function, this will prevent it from being overwritten.

```
const log = message => console.log(message)
```

Since functions are variables, we can add them to objects.

```
const obj = {
    message: "They can be added to objects like variables",
    log(message) {
        console.log(message)
    }
}

obj.log(obj.message)

// They can be added to objects like variables
```

We can also add functions to arrays in JavaScript.

```
const messages = [
    "They can be inserted into arrays",
    message => console.log(message),
    "like variables",
    message => console.log(message)
]

messages[1](messages[0])   // They can be inserted into arrays
messages[3](messages[2])   // like variables
```

Functions can be sent to other functions as arguments just like other variables.

```
const insideFn = logger =>
  logger("They can be sent to other functions as arguments");
```

```
insideFn(message => console.log(message))

// They can be sent to other functions as arguments
```

They can also be returned from other functions... just like variables

```
var createScream = function(logger) {
    return function(message) {
        logger(message.toUpperCase() + "!!!")
    }
}

const scream = createScream(message => console.log(message))

scream('functions can be returned from other functions')
scream('createScream returns a function')
scream('scream invokes that returned function')

// FUNCTIONS CAN BE RETURNED FROM OTHER FUNCTIONS!!!
// CREATESCREAM RETURNS A FUNCTION!!!
// SCREAM INVOKES THAT RETURNED FUNCTION!!!
```

The last two examples were of higher order functions, functions that either take or return other functions. Using ES6 syntax, we could describe the same createScream higher order function with arrows.

```
const createScream = logger => message =>
    logger(message.toUpperCase() + "!!!")
```

From here on out, we need to pay attention to the number of arrows used during function declaration. More than one arrow means that we have a higher order function.

We can say that JavaScript is a functional language because its functions are first class citizens. This means that functions are data. They can be saved, retrieved, or flow through your applications just like variables.

Imperative vs Declarative

Functional programming is a part of a larger programming paradigm: declarative programming. Declarative programming is a style of programming where applications are structured in a way that prioritizes describing what should happen over defining how it should happen.

In order to understand declarative programming, we'll contrast it with imperative programming, or a style of programming that is only concerned with how to achieve results with code. Let's consider a common task: making a string URL friendly. Typically, this can be accomplished by replacing all of the spaces in a string with hyphens, since spaces are not URL friendly. First, let's examine an imperative approach to this task.

```
var string = "This is the mid day show with Cheryl Waters";
var urlFriendly = "";

for (var i=0; i<string.length; i++) {
  if (string[i] === " ") {
    urlFriendly += "-";
  } else {
    urlFriendly += string[i];
  }
}

console.log(urlFriendly);
```

In this example, we loop through every character in the string replacing spaces as they occur. The structure of this program is only concerned with how such a task can be achieved. We use a for loop, an if statement, and set values with an equal operator. Just looking at the code alone does not tell us much. Imperative programs require lots of comments in order to understand what is going on.

Now let's look at a declarative approach to the same problem.

```
const string = "This is the mid day show with Cheryl Waters"
const urlFriendly = string.replace(/ /g, "-")

console.log(urlFriendly)
```

Here we are using string.replace along with a regular expression to replace all instances of spaces with hyphens. Using string.replace is a way of describing what is supposed to happen, spaces in the string should be replaced. The details of how spaces are dealt with are abstracted away inside the .replace function. In a declarative program, the syntax itself describes what should happen and abstract away the details of how things happen through abstraction.

Declarative programs are easy to reason about because the code itself describes what is happening. For example, read the syntax in the following sample, it details what happens after members are loaded from an API.

```
const loadAndMapMembers = compose(
    combineWith(sessionStorage, "members"),
    save(sessionStorage, "members"),
    scopeMembers(window),
    logMemberInfoToConsole,
    logFieldsToConsole("name.first"),
    countMembersBy("location.state"),
    prepStatesForMapping,
    save(sessionStorage, "map"),
    renderUSMap
);
```

```
getFakeMembers(100).then(loadAndMapMembers);
```

The declarative approach is more readable and, thus, easier to reason about. The details of how each of these functions is implemented are abstracted away. Those tiny functions are named well and combined in a way that describes how member data goes from being loaded to being saved and printed on a map. This approach does not require many comments. Declarative programming should produce applications that are easier to reason about. When it is easier to reason about an application, that application is easier to scale[5] .

Now, let's consider the task of building a DOM. An imperative approach would be concerned with how the DOM is constructed.

```
var target = document.getElementById('target');
var wrapper = document.createElement('div');
var headline = document.createElement('h1');

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

This code is concerned with creating elements, setting elements, and adding them to the document. It would be very hard to make changes, add features, or scale 10,000 lines of code where the DOM is constructed imperatively.

Now let's take a look at how we can construct a DOM declaratively using a React component.

```
const { render } = ReactDOM

const Welcome = () => (
    <div id="welcome">
        <h1>Hello World</h1>
    </div>
)

render(
    <Welcome />,
    document.getElementById('target')
)
```

React is declarative. Here, the welcome component describes the DOM that should be rendered. The render function uses the instructions declared in the component to build the DOM. The render function abstracts away the details of how the DOM is to

5 Additional detail about the declarative programming paradigm can be found here: http://c2.com/cgi/wiki?DeclarativeProgramming

be rendered. We can clearly see that we want to render our welcome component into the element with the id of "target".

Functional Concepts

Now that we have been introduced to functional programming, and what it means to be "functional" or "declarative", we will move on to introducing the core concepts of functional programming: immutability, purity, data abstraction, higher-order functions, recursion, and composition.

Immutability

To mutate is to change, so to be immutable is to be unchangeable. In a functional program, data is immutable, it never changes.

If you needed to share your birth certificate with the public, but wanted to redact or remove private information you essentially have two choices: you can take a big Sharpie to your original birth certificate and cross out private data, or you can find a copy machine. Finding a copy machine, making a copy of your birth certificate, and writing all over that copy with that big Sharpie would be preferable. This way you can have a redacted birth certificate, which you can share and your original which is still intact.

This is how immutable data works in an application. We will not change the original data structures. We will build changed copies of those data structures and use them instead.

To understand how immutability works, let's take a look at what it means to mutate data. Consider an object that represents the color lawn.

```
let color_lawn = {
    title: "lawn",
    color: "#00FF00",
    rating: 0
}
```

We could build a function that would rate colors, and use that function to change the rating of the color object.

```
function rateColor(color, rating) {
  color.rating = rating
  return color
}

console.log(rateColor(color_lawn, 5).rating)     // 5
console.log(color_lawn.rating)                   // 5
```

In JavaScript, function arguments are references to the actual data. Setting the color's rating would change or mutate the original color object. Imagine if you tasked a business with redacting and sharing your birth certificate and they returned your original birth certificate with black marker covering the important details. This rate color function is bad for business because it changing the original color.

You would hope that a business would use enough common sense to make a copy of your birth certificate and return the original unharmed. We can rewrite the rateColor function so that it does not harm the original goods, the color object.

```
var rateColor = function(color, rating) {
   return Object.assign({}, color, {rating:rating})
}

console.log(rateColor(color_lawn, 5).rating)      // 5
console.log(color_lawn.rating)                    // 4
```

Here, we used Object.assign to change the color rating. Object.assign is the copy machine. It takes a blank object, copies the color to that object, and overwrites the rating on the copy. Now we can have a newly rated color object without having to change the original.

We can write the same function using an ES6 arrow function along with the ES7 object spread operator. This rate color function uses the spread operator to copy the color into a new object and then overwrite its rating.

```
const rateColor = (color, rating) =>
    ({
        ...color,
        rating
    })
```

This emerging JavaScript version of the rateColor is exactly the same as the previous. It treats the color as an immutable object. It just does so with less syntax and looks a little bit cleaner. Notice that we wrap the returned object in parentheses. With arrow functions, this is a required step since the arrow can't just point to an object's curly braces.

Let's consider an array of color names.

```
let list = [
    { title: "Rad Red"},
    { title: "Lawn"},
    { title: "Party Pink"}
]
```

We could create a function that will add colors to that array using array.push().

```
var addColor = function(title, colors) {
  colors.push({ title: title })
  return colors;
}

console.log(addColor("Glam Green", list).length)        // 4
console.log(list.length)                                // 4
```

However, array.push() is not immutable. This addColor function changes the original array by adding another field to it. In order to keep the colors array immutable, we must use array.concat instead.

```
const addColor = (title, array) => array.concat({title})

console.log(addColor("Glam Green", list).length)        // 4
console.log(list.length)                                // 3
```

Array.concat concatenates arrays. In this case, it takes a new object, with a new color title, and adds it to a copy of the original array.

You can also use the ES6 spread operator to concatenate arrays in the same way it can be used to copy objects. Here is the emerging JavaScript equivalent of the previous addColor function.

```
const addColor = (title, list) => [...list, {title}]
```

This function copies the original list to a new array and then adds a new object containing the color's title to that copy. It is immutable.

Pure Functions

A pure function is a function that returns a value that is computed based on its arguments. Pure functions take at least one argument and always return a value or another function. They do not cause side effects. They do not set global variables or change anything about application state. They treat their arguments as immutable data. If you send a specific argumentATLAS-CURSOR-HERE to a pure function, you can expect a specific result.

In order to understand pure functions, we will first take a look at an impure function.

```
var frederick = {
    name: "Frederick Douglass",
    canRead: false,
    carWrite: false
}

function selfEducate() {
    frederick.canRead = true
    frederick.canWrite = true
    return frederick
}

selfEducate()
```

```
console.log( frederick )

// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

The selfEducate function is not a pure function. It does not take any arguments, and it does not return a value or a function. It also changes a variable outside of its scope: frederick. Once the selfEducate function is invoked, something about the "world" has changed. It causes side effects.

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
}

const selfEducate = (person) => {
  person.canRead = true
  person.canWrite = true
  return person
}

console.log( selfEducate(frederick) )
console.log( frederick )

// {name: "Frederick Douglass", canRead: false, canWrite: false}
// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

Pure functions are testable

Pure functions are naturally testable. They do not change anything about their environment or "world", and therefore do not require a complicated test setup or teardown. Everything a pure function needs to operate it accesses via arguments. When testing a pure function, you control the arguments, and thus you can estimate the outcome. More on testing in Chapter 11.

This selfEducate function is also impure. It causes side effects. Invoking this selfEducate function mutates the objects that are sent to it. If we can treat the arguments sent to this function as immutable data, then we would have ourselves a pure function.

Let's have this function take in an argument.

```
const frederick = {
    name: "Frederick Douglass",
    canRead: false,
    canWrite: false
}

const selfEducate = person =>
    ({
        ...person,
        canRead: true,
        canWrite: true
    })

console.log( selfEducate(frederick) )
console.log( frederick )

// {name: "Frederick Douglass", canRead: false, canWrite: false}
// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

Finally, this version of the selfEducate is a pure function. It computes a value based on the argument that was sent to it - the person. It returns a new person object without mutating the argument sent to it and therefore has no side effects.

Now let's examine an impure function that mutates the DOM.

```
function Header(text) {
  let h1 = document.createElement('h1');
  h1.innerText = text;
  document.body.appendChild(h1);
}

Header("Header() caused side effects");
```

The header function creates a heading one element with specific text and adds it to the DOM. This function is impure. It does not return a function or a value, and it causes side effects: a changed DOM.

In React, UI is expressed with pure functions. In this sample, Header is a pure function that can be used to create heading one elements just like in the previous example. However this function on its own does not cause side effects because it does not mutate the DOM. This function will create a heading one element, and it is up to some other part of the application to use that element to change the DOM.

```
const Header = (props) => <h1>{props.title}</h1>
```

Pure functions are another core concept of functional programming. They will make your life much easier because they will not affect your application's state. When writing functions try to follow these three rules:

1. The function should take in at least one argument
2. The function should return a value or another function
3. The function should not change or mutate any of its arguments

Data Transformations

How does anything change in an application if the data is immutable? Functional programming is all about transforming data from one form to another. We will produce transformed copies using functions. These functions make our code less imperative and thus reduce complexity.

> In order to use React, you have to learn how to use JavaScript.
> - Dan Abramov

Much of functional programming is about transforming data. You do not need a special framework to understand how produce one dataset that is based upon another. JavaScript already has the necessary tools for this task built into the language. There are two core functions that you must master in order to be proficient with functional JavaScript. These functions are array.map and array.reduce.

In this section, we will take a look at how we can use some of these core functions to transform data from one type to another.

Consider this array of high schools.

```
const schools = [
  "Yorktown",
  "Washington & Lee",
  "Wakefield"
]
```

We can get a comma delimited list of these strings by using the array.join function.

```
console.log( schools.join(", ") )

// "Yorktown, Washington & Lee, Wakefield"
```

The join function is a built-in JavaScript array method that we can use to extract a delimited string from our array. The original array is still intact. Join simply provides a different take on it. The details of how this string is produced are abstracted away from the programmer.

If we wanted to create a function that creates a new array of the schools that begin with the letter "W" we could use the array.filter method.

```
const wSchools = schools.filter(school => school[0] === "W")

console.log( wSchools )
// ["Washington & Lee", "Wakefield"]
```

Arrray.filter is a built-in JavaScript function that produces a new array from a source array. This function takes a *predicate* as its only argument. A predicate is a function that always returns a boolean value: true or false. Array.filter invokes this predicate once for every item in the array. That item is passed to the predicate as an argument and used to decide if that item shall be added to the new array. In this case, array.filter is checking every school to see if it begins with a "W".

When it is time to remove an item from an array we should use array.filter over array.pop() or array.splice() because array.filter is immutable. In this next sample, the cutSchool function returns new arrays that filter out specific school names.

```
const cutSchool = (cut, list) =>
    list.filter(school => school !== cut)

console.log(cutSchool("Washington & Lee", schools).join(" * "))

// "Yorktown * Wakefield"

console.log(schools.join("\n"))

// Yorktown
// Washington & Lee
// Wakefield
```

In this case, the cutSchool function was used to return a new array that does not contain "Washington & Lee". Then the join function is used with this new array to create a star delimited string out of the remaining two schools. CutSchool is a pure function. It takes a list of schools and the name of the school that should be removed and returns the new array without that specific school. Additionally, the join function has been chained on to produce a star delimited string out of the returned array.

Another array function that is essential to functional programming is array.map. Instead of a predicate, the array.map method takes a function as its argument. This function will be invoked once for every item in the array, and whatever it returns will be added to the new array.

```
const highSchools = schools.map(school => `${school} High School`)

console.log(highSchools.join("\n"))

//  Yorktown High School
//  Washington & Lee High School
//  Wakefield High School

console.log(schools.join("\n"))

//  Yorktown
//  Washington & Lee
//  Wakefield
```

In this case, the map function was used to append "High School" to each school name. The schools array is still intact.

In the last example, we produced an array of strings from an array of strings. The map function could produce an array of objects, values, arrays, other functions - any JavaScript type. Here is an example of the map function returning an object for every school.

```
const highSchools = schools.map(school => ({ name: school }))

console.log( highSchools )

// [
//   { name: "Yorktown" },
//   { name: "Washington & Lee" },
//   { name: "Wakefield" }
// ]
```

An array containing objects was produced from an array that contains strings.

If you need to create a pure function that changes one object in an array of objects, the map function can be used. In the following example, we will change the school with the name of Stratford to HB Woodlawn without mutating the schools array.

```
let schools = [
    { name: "Yorktown"},
    { name: "Stratford" },
    { name: "Washington & Lee"},
    { name: "Wakefield"}
]

let updatedSchools = editName("Stratford", "HB Woodlawn", schools)

console.log( updatedSchools[1] )  // { name: "HB Woodlawn" }
console.log( schools[1] )         // { name: "Stratford" },
```

The schools array is an array of objects. The updatedSchools variable calls the edit-Name function and we send it the school we want to update, the new school, and the schools array. This changes the new array but makes no edits to the original.

```
const editName = (oldName, name, arr) =>
    arr.map(item => {
        if (item.name === oldName) {
            return {
                ...item,
                name
            }
        } else {
            return item
        }
    })
```

Within editName, the map function is used to create a new array of objects based upon the original array. Array.map injects the index of each item into the callback as the second argument, the variable i. When i is not equal to the index of the item we wish to edit, we'll simply package the same item into the new array. When i is equal to the index of the item that we wish to edit, we replace the item at that index in the new array with a new object.

The editName function can be written entirely in one line. The is an example of the same function using a shorthand if/else statement

```
const editName = (oldName, name, arr) =>
    arr.map(item => (item.name === oldName) ?
        ({...item,name}) :
        item
    )
```

If you needed to transform an array into an object, you can use array.map in conjunction with Object.keys. Object.keys is a method that can be used to return an array of keys from an object.

Let's say we needed to transform an array of school objects from a hash of schools.

```
const schools = {
  "Yorktown": 10,
  "Washington & Lee": 2,
  "Wakefield": 5
}

const schoolArray = Object.keys(schools).map(key =>
        ({
            name: key,
            wins: schools[key]
        })
    )

console.log(schoolArray)

// [
//   {
//     name: "Yorktown",
//     wins: 10
//   },
//   {
//     name: "Washington & Lee",
//     wins: 2
//   },
//   {
//     name: "Wakefield",
//     wins: 5
//   }
// ]
```

In this example, Object.keys returns an array of school names, and we can use map on that array to produce a new array of the same length. The name of the new object will be set using the key, and the wins is set equal to the value.

So far we've learned that we can transform arrays with array.map and array.filter. We've also learned that we can change arrays into objects by combining Object.keys with Array.map. The final tool that that we need in our functional arsenal is the ability to transform arrays into primitives and other objects.

The reduce and reduceRight function can be used to transform an array into any value. Any value means a number, string, boolean, object, or even function.

Let's say we needed to find the maximum number in an array of numbers. We need to transform an array into a number; therefore, we can use reduce.

```
const ages = [21,18,42,40,64,63,34];

const maxAge = ages.reduce((max, age) => {
    console.log(`${age} > ${max} = ${age > max}`);
    if (age > max) {
        return age
    } else {
        return max
    }
}, 0)

console.log('maxAge', maxAge);

// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

The ages array has been reduced into a single value: the maximum age: 64. Reduce takes two arguments: a callback function and an original value. In this case, the original value is 0, which sets the initial maximum value to 0. The callback is invoked once for every item in the array. The first time this callback is invoked the age is equal to 21, the first value in the array, and max is equal to 0, the initial value. The callback returns the greater of the two numbers, 21, and that becomes the max value during the next iteration. Each iteration compares each age against the max value and returns the greater of the two. Finally, the last number in the array is compared and returned from the previous callback.

If we remove the console.log statement from the above function and use a shorthand if/else statement, we can calculate the max value in any array of numbers with the following syntax:

```
const max = ages.reduce(
    (max, value) => (value > max) ? value : max,
    0
)
```

array.reduceRight

Array.reduce right works the same way as array.reduce, the difference is that it starts reducing from the end of the array rather than the beginning.

Sometimes we need to transform an array into an object. The following example uses reduce to transform an array that contains colors into a hash.

```
const colors = [
    {
        id: '-xekare',
        title: "rad red",
        rating: 3
    },
    {
        id: '-jbwsof',
        title: "big blue",
        rating: 2
    },
    {
        id: '-prigbj',
        title: "grizzly grey",
        rating: 5
    },
    {
        id: '-ryhbhsl',
        title: "banana",
        rating: 1
    }
]

const hashColors = colors.reduce(
    (hash, {id, title, rating}) => {
        hash[id] = {title, rating}
        return hash
    },
    {}
)

console.log(hashColors);

// {
//   "-xekare": {
//     title:"rad red",
//     rating:3
//   },
//   "-jbwsof": {
//     title:"big blue",
//     rating:2
//   },
//   "-prigbj": {
```

```
//     title:"grizzly grey",
//     rating:5
//   },
//   "-ryhbhsl": {
//     title:"banana",
//     rating:1
//   }
// }
```

In the above example, the second argument sent to the reduce function is an empty object. This is our initial value for hash. During each iteration, the callback function adds a new key to the hash using bracket notation and sets the value for that key to the id field of the array. Array.reduce reduces can be used to reduce an array to a single value, in this case, an object.

We can even transform arrays into completely different arrays using reduce. There are cases where array.reduce is a better choice. Consider reducing an array with multiple instances of the same value to an array of distinct values. The reduce method can be used to accomplish this task.

```
const colors = ["red", "red", "green", "blue", "green"];

const distinctColors = colors.reduce(
    (distinct, color) =>
        (distinct.indexOf(color) !== -1) ?
            distinct :
            [...distinct, color],
    []
)

console.log(distinctColors)

// ["red", "green", "blue"]
```

In this example, the colors array is reduced to an array of distinct values. The second argument sent to the reduce function is an empty array. This will be the initial value for distinct. When the distinct array does not already contain a specific color, it will be added. Otherwise, it will be skipped, and the current distinct array will be returned.

Map and reduce are the main weapons of any functional programmer, and JavaScript is no exception. If you want to be a proficient JavaScript engineer, then you must master these functions. The ability to create one data set from another is a required skill and is useful for any type of programming paradigm.

Higher Order Functions

The use of higher order functions is also essential to functional programming. We've already mentioned higher order functions several times over, and we've even used a few in this chapter. Higher order functions are functions that can manipulate other functions. They can either take functions in as arguments or return functions or both.

The first category of higher order functions are functions that expect other functions as arguments. Array.map, array.filter, and array.reduce all take functions as arguments. They are higher order functions.[6]

Let's take a look at how we can implement a higher order function. In the following example, we will create an invokeIf callback function that will test a condition and invoke a callback function when it is true and another callback function when that condition is false.

```
const invokeIf = (condition, fnTrue, fnFalse) =>
    (condition) ? fnTrue() : fnFalse()

const showWelcome = () =>
    console.log("Welcome!!!")

const showUnauthorized = () =>
    console.log("Unauthorized!!!")

invokeIf(true, showWelcome, showUnauthorized)    // "Welcome"
invokeIf(false, showWelcome, showUnauthorized)   // "Unauthorized"
```

InvokeIf expects two functions: one for true, and one for false. This is demonstrated by sending both showWelcome and showUnauthorized to invokeIf. When the condition is true, showWelcome is invoked. When it is false, showUnauthorized is invoked.

Higher order functions that return other functions can help us handle the complexities associated with asynchronicity in JavaScript. They can help us create functions that can be used or reused at our convenience.

Currying is a functional technique that involves the use of higher order functions. Currying is the practice of holding on to some of the values needed to complete an operation until the rest can be supplied at a later point in time. This is achieved through the use of a function that returns another function, the curried function.

The following is an example of currying. The userLogs function hangs on to some information - the user name - and returns a function that can be used and reused when the rest of the information - the message - is made available. In this example,

6 For more on higher-order functions, check out Eloquent JavaScript, Chapter 5. http://eloquentjavascript.net/05_higher_order.html

log messages will all be prepended with the associated username. Notice that we're using the getFakeMembers function that returns a promise from chapter 2.

```
const userLogs = userName => message =>
    console.log(`${userName} -> ${message}`)

const log = userLogs("grandpa23")

log("attempted to load 20 fake members")
getFakeMembers(20).then(
    members => log(`successfully loaded ${members.length} members`),
    error => log("encountered an error loading members")
)

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> successfully loaded 20 members

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> encountered an error loading members
```

UserLogs is the higher order function. The log function is produced from userLogs, and every time the log function is used, "grandpa23" is prepended to the message.

Recursion

Recursion is a technique that involves creating functions that recall themselves. Often when faced with a challenge that involves a loop, a recursive function can be used instead. Consider the task of counting down from 10. We could create a for loop to solve this problem, or we could alternatively use a recursive function. In this example, countdown is the recursive function.

```
const countdown = (value, fn) => {
    fn(value)
    return (value > 0) ? countdown(value-1, fn) : value
}

countdown(10, value => console.log(value));

// 10
// 9
// 8
// 7
// 6
// 5
// 4
// 3
// 2
// 1
// 0
```

Countdown expects a number and a function as arguments. In this example, countdown is invoked with a value of 10 and a callback function. When countdown is invoked, the callback is invoked which logs the current value. Next, countdown checks the value to see if it is greater than 0. If it is, countdown recalls itself with a decremented value. Eventually, the value will be 0 and countdown will return that value all the way back up the call stack.

Browser Call Stack Limitations

Recursion should be used over loops wherever possible, but not all JavaScript engines are optimized for a large amount of recursion. Too much recursion can cause JavaScript errors. These errors can be avoided by implementing advanced techniques to clear the call stack and flatten out recursive calls. Future JavaScript engines plan eliminate any call stack limitations entirely..

Recursion is another functional technique that works well with asynchronous processes. Functions can recall themselves when they are ready.

The countdown function can be modified to count down with a delay. This modified version of the countdown function can be used to create a countdown clock.

```
const countdown = (value, fn, delay=1000) => {
    fn(value)
    return (value > 0) ?
        setTimeout(() => countdown(value-1, fn), delay) :
        value
}

const log = value => console.log(value)
countdown(10, log);
```

In this example, we create a 10-second countdown by initially invoking countdown once with the number 10 in a function that logs the countdown. Instead of recalling itself right away, the countdown function waits one second before recalling itself, thus creating a clock.

Recursion is a good technique for searching data structures. You can use recursion to iterate through subfolders until the folder, the one that contains only files, is identified. You can use recursion to iterate though the HTML DOM until you find the one that does not contain any children. In the next example, we will use recursion to iterate deeply into an object to retrieve a nested value.

```
var dan = {
    type: "person",
    data: {
      gender: "male",
      info: {
        id: 22,
        fullname: {
          first: "Dan",
          last: "Deacon"
        }
      }
    }
  }

deepPick("type", dan);                      // "person"
deepPick("data.info.fullname.first", dan);  // "Deacon"
```

DeepPick, can be used to access dan's type, stored immediately in the first object, or dig down into nested objects to locate dan's first name. Sending a string that uses dot notation, we can specify where to locate values that are nested deep within an object.

```
const deepPick = (fields, object={}) => {
   const [first, ...remaining] = fields.split(".")
   return (remaining.length) ?
       deepPick(remaining.join("."), object[first]) :
       object[first]
}
```

The deepPick function is either going to return a value or recall itself, until it eventually returns a value. First, this function splits the dot notated fields string into an array and uses array destructuring to separate the first value from the remaining values. If there are remaining values, deepPick recalls itself with slightly different data, allowing it to dig one level deeper.

This function continues to call itself until the field string no longer contains dots, meaning that there are no more remaining fields. In this sample, you can see how the values for first, remaining, and object[first] change as deepPick iterates through match.

```
deepPick("data.info.fullname.first", dan);  // "Deacon"

// First Iteration
//   first = "data"
//   remaining.join(".") = "info.fullname.first"
//   object[first] = { gender: "male", {info} }

// Second Iteration
//   first = "info"
//   remaining.join(".") = "fullname.first"
//   object[first] = {id: 22, {fullname}}

// Third Iteration
//   first = "fullname"
//   remaining.join("." = "first"
//   object[first] = {first: "Dan", last: "Deacon" }

// Finally...
```

```
    //   first = "first"
    //   remaining.length = 0
    //   object[first] = "Deacon"
```

Recursion is a powerful functional technique that is fun to implement. Use recursion over looping whenever possible.

Composition

Functional programs break their logic up into small pure functions that are focused on specific tasks. Eventually, you will need to put these smaller functions together. Specifically, you may need to combine them, call them in series or parallel, or compose them into larger functions until you eventually have an application.

When it comes to composition, there are a number of different implementations, patterns, and techniques. One that you may be familiar with is chaining. In JavaScript functions can be chained together using dot notation to act on the return value of the previous function.

Strings have a replace method. The replace method returns a template string which also will have a replace method. Therefore, we can chain together replace methods with dot notation to transform a string.

```
const template = "hh:mm:ss tt"
const clockTime = template.replace("hh", "03")
      .replace("mm", "33")
      .replace("ss", "33")
      .replace("tt", "PM")

console.log(clockTime)

// "03:33:33 PM"
```

In this example, the template is a string. By chaining replace methods to the end of the template string, we can replace hours, minutes, seconds, and time of day in the string with new values. The template itself remain intact and can be reused to create more clock time displays.

 Chaining is one composition technique, but there are others. The goal of composition is to "generate a higher order function by combining simpler functions."[7]

```
const both = date => appendAMPM(civilianHours(date))
```

The both function is one function that pipes a value through two separate functions. The output of civilian hours becomes the input for appendAMPM, and we can

7 Functional.js Composition, http://functionaljs.com/functions/compose/

change a date using both of these functions combined into one. However, this syntax is hard to comprehend and therefore tough to maintain or scale. What happens when we need to send a value through 20 different functions?

A more elegant approach is to create a higher order function that we can use to compose functions into larger functions.

```
const both = compose(
    civilianHours,
    appendAMPM
)

both(new Date())
```

This approach looks much better. It is easy to scale because we can add more functions at any point. This approach also makes it easy to change the order of the composed functions.

The compose function is a higher order function. It takes functions as arguments and returns a single value.

```
const compose = (...fns) =>
  (arg) =>
    fns.reduce(
      (composed, f) => f(composed),
      arg
    )
```

Compose takes in functions as arguments and returns a single function. In this implementation, the spread operator is used to turn those function arguments into an array called fns. A function is then returned that expects one argument, arg. When this function is invoked, the fns array is piped starting with the argument we want to send through the function. The argument becomes the initial value for composed and then each iteration of the reduced callback returns. Notice that the callback takes two arguments: composed and a function f. Each function is invoked with compose which is the result of the previous functions output. Eventually, the last function will be invoked and the last result returned.

This is a simple example of a compose function designed to illustrate composition techniques. This function becomes more complex when it is time to handle more than one argument or deal with arguments that are not functions. Other implementations of compose [8] may use reduceRight which would compose the functions in reverse order.

8 Another implementation of compose is found in Redux: http://redux.js.org/docs/api/compose.html

Putting it all together

Now that we've been introduced to the core concepts of functional programming, let's put those concepts to work for us and build a small JavaScript application.

Since JavaScript will let you slip away from the functional paradigm, and you do not have to follow the rules, you will need to stay focused. Following these three simple rules will help you stay on target.

1. Keep Data Immutable
2. Keep Functions Pure : accept at least one argument, return data or another function
3. Use Recursion over looping (wherever possible)

Our challenge is to build a ticking clock. The clock needs to display hours, minutes, seconds and time of day in civilian time. Each field must always have double digits, that means leading zeros need to be applied to single digit values like 1 or 2. The clock must also tick and change the display every second.

First, let's review an imperative solution for the clock.

```
// Log Clock Time every Second
setInterval(logClockTime, 1000);

function logClockTime() {

  // Get Time string as civilian time
  var time = getClockTime();

  // Clear the Console and log the time
  console.clear();
  console.log(time);
}

function getClockTime() {

  // Get the Current Time
  var date = new Date();
  var time = "";

  // Serialize clock time
  var time = {
    hours: date.getHours(),
    minutes: date.getMinutes(),
    seconds: date.getSeconds(),
    ampm: "AM"
  }
```

```
        // Convert to civilian time
        if (time.hours == 12) {
          time.ampm = "PM";
        } else if (time.hours > 12) {
          time.ampm = "PM";
          time.hours -= 12;
        }

        // Prepend a 0 on the hours to make double digits
        if (time.hours < 10) {
          time.hours = "0" + time.hours;
        }

        // prepend a 0 on the minutes to make double digits
        if (time.minutes < 10) {
          time.minutes = "0" + time.minutes;
        }

        // prepend a 0 on the seconds to make double digits
        if (time.seconds < 10) {
          time.seconds = "0" + time.seconds;
        }

        // Format the clock time as a string "hh:mm:ss tt"
        return time.hours + ":"
             + time.minutes + ":"
             + time.seconds + " "
             + time.ampm;

    }
```

This solution is pretty straight forward. It works, the comments help us understand what is happening. However, these functions are large and complicated. Each function does a lot. They are hard to comprehend, they require comments and they are tough to maintain. Let's see how a functional approach can produce a more scalable application.

Our goal will be to break the application logic up into smaller parts, functions. Each function will be focused on a single task, and we will compose them into larger functions that we can use to create the clock.

First, lets create some functions that give us values and manage the console. We'll need a function that gives us one second, a function that gives us the current time, and a couple of functions that will log messages on a console and clear the console. In functional programs, we should use functions over values wherever possible. We will invoke the function to obtain the value when needed.

```
const oneSecond = () => 1000
const getCurrentTime = () => new Date()
const clear = () => console.clear()
const log = message => console.log(message)
```

Next we will need some functions for transforming data. These three functions will be used to mutate the Date object into an object that can be used for our clock.

serializeClockTime
: Takes a date object and returns a object for clock time that contains hours minutes and seconds.

civilianHours
: Takes the clock time object and returns an object where hours are converted to civilian time. For example: 1300 becomes 1 o'clock

appendAMPM
: Takes the clock time object and appends time of day, AM or PM, to that object.

```
const serializeClockTime = date =>
    ({
        hours: date.getHours(),
        minutes: date.getMinutes(),
        seconds: date.getSeconds()
    })

const civilianHours = clockTime =>
    ({
        ...clockTime,
        hours: (clockTime.hours > 12) ?
            clockTime.hours - 12 :
            clockTime.hours
    })

const appendAMPM = clockTime =>
    ({
        ...clockTime,
        ampm: (clockTime.hours >= 12) ? "PM" : "AM"
    })
```

These three functions are used to transform data without changing the original. They treat their arguments as immutable objects.

Next we'll need a few higher order functions.

display
: Takes a target function and returns a function that will send a time to the target. In this example the target will be console.log.

formatClock
: Takes a template string and uses it to return clock time formatted based upon the criteria from the string. In this example, the template is "hh:mm:ss tt". From ther, formatClock will replaces the placeholders with hours, minutes, seconds, and time of day.

prependZero
: Takes an object's key as an argument and prepends a zero to the value stored under that objects key. It takes in a key to a specific field and prepends values with a zero if the value is less than 10.

```
const display = target => time => target(time)

const formatClock = format =>
    time =>
        format.replace("hh", time.hours)
              .replace("mm", time.minutes)
              .replace("ss", time.seconds)
              .replace("tt", time.ampm)

const prependZero = key => clockTime =>
    ({
        ...clockTime,
        [key]: (clockTime[key] < 10) ?
            "0" + clockTime[key] :
            clockTime[key]
    })
```

These higher order functions will be invoked to create the functions that will be reused to format the clock time for every tick. Both format clock and prependZero will be invoked once, initially setting up the required template or key. The inner functions that they return will be invoked once every second to format the time for display.

Now that we have all of the functions required to build a ticking clock, we will need to compose them. We will use the compose function that we defined in the last section to handle composition.

convertToCivilianTime
: A single function that will take clock time as an argument and transforms it into civilian time by using both civilian hours.

doubleDigits
: A single function that will take civilian clock time and make sure the hours, minutes, and seconds display double digits by prepending zeros where needed.

startTicking
: Starts the clock by setting an interval that will invoke a callback every second. The callback is composed using all of our functions. Every second the console is cleared, currentTime obtained, converted, civilianized, formatted, and displayed.

```
const convertToCivilianTime = clockTime =>
    compose(
        appendAMPM,
        civilianHours
    )(clockTime)
```

```
const doubleDigits = civilianTime =>
    compose(
        prependZero("hours"),
        prependZero("minutes"),
        prependZero("seconds")
    )(civilianTime)

const startTicking = () =>
    setInterval(
        compose(
            clear,
            getCurrentTime,
            serializeClockTime,
            convertToCivilianTime,
            doubleDigits,
            formatClock("hh:mm:ss tt"),
            display(log)
        ),
        oneSecond()
    )

startTicking()
```

This declarative version of the clock achieves the same results as the imperative version. However, there quite a few benefits to this approach. First, all of these functions are easily testable and reusable. They can be used in future clocks or other digital displays. Also, this program is easily scalable. There are no side effects. There are no global variables outside of functions themselves. There could still be bugs, but they will be easier to find.

In this chapter, we've introduced functional programming principles. Throughout the book when we discuss best practices in React and Flux, we'll demonstrate how these libraries are based in functional techniques. In the next chapter, we will dive into React officially, with an improved understanding of the principles that guided its development.